Aalto University
School of Electrical
Engineering

Tero Takala

Model-driven application integration for manufacturing execution systems

Thesis submitted for examination.

Espoo 28.5.2018
Thesis supervisor: Professor Valeriy Vyatkin
Thesis advisor: D.Sc. Ilkka Seilonen

Tekijä Tero Takala		
Työn nimi Mallipohjainen järjestelmäintegraatio tuotannonohjausjärjestelmille		
Koulutusohjelma Master's Programme in Automation and Electrical Engineering		
Pää-/sivuaine Control, Robotics and Autonomous Systems		**Koodi** ELEC3025
Työn valvoja Professor Valeriy Vyatkin		
Työn ohjaaja(t) D.Sc. Ilkka Seilonen		
Päivämäärä 28.5.2018	**Sivumäärä** 39	**Kieli** Englanti

Tiivistelmä

Järjestelmäintegraatio vaikeutuu ohjelmien monimutkaistuessa. Tässä työssä tutkitaan mallipohjaisten järjestelmäintegraatiometodien soveltuvuutta tuotannonohjausjärjestelmille (MES). Tavoitteena oli muodostaa koodigeneraattori, joka käyttää malleja luodakseen toimivan ohjelman, joka siirtää tietoa MES-järjestelmästä johonkin toiseen tietojärjestelmään. Toteutuksessa keskityttiin yleistettävyyteen.

Aluksi työssä käytiin läpi aikaisempaa tutkimusta MES-järjestelmistä ja mahdollisuuksista integroida niitä toisiin informaatiojärjestelmiin. Lisäksi otettiin selvää kansainvälisestä ISA-95 standardista ja B2MML:sta sekä mallipohjaisesta tekniikasta (MDE). Tämän jälkeen järjestelmälle määriteltiin vaatimukset, jotka jaettiin käyttäjän ja kehittäjän vaatimuksiin. Koodigeneraattorista tehtiin ehdot täyttävä suunnitelma, joka toteutettiin ja jolla suoritettiin kokeita. Koe toteutettiin lukemalla tuotantodataa MES:n kaltaisen Delfoi Plannerin tietokannasta, jonka jälkeen data muutettiin B2MML tyyliä noudattavaan XML-schema muotoon.

Kokeet osoittivat, että koodigeneraattori toimi kuten toivottiin. Kuitenkin havaittiin, että verrattuna manuaalisesti toteutettuun ohjelmaan, luotu ohjelma ei ollut yhtä tehokas ja lisäksi se oli pidempi. Huomattiin myös, että MDE-metodien käyttöönotto vie paljon aikaa. Jotta MDE olisi perinteistä ohjelmointia parempi vaihtoehto, sitä pitäisi käyttää useita kertoja ja sillä luotu järjestelmä ei saisi olla liian aikariippuvainen. Havaintojen perusteella voidaan sanoa, että mallipohjaisia järjestelmäintegraatiometodeja voidaan käyttää MES-järjestelmien integrointiin, mutta sille on rajoituksia.

Avainsanat MDE, software integration, MES, ERP, ISA-95, B2MML

Author Tero Takala		
Title of thesis Model-driven application integration for manufacturing execution systems		
Degree programme Master's Programme in Automation and Electrical Engineering		
Major/minor Control, Robotics and Autonomous Systems		
Thesis supervisor Professor Valeriy Vyatkin		
Thesis advisor(s) D.Sc. Ilkka Seilonen		
Date 28.5.2018	**Number of pages** 39	**Language** English

Abstract

Application integration becomes more complex as software becomes more advanced. This thesis investigates the applicability of model-driven application integration methods to the software integration of manufacturing execution systems (MES). The goal was to create a code generator that uses models to generate a working program that transfers data from a MES to another information system. The focus of the implementation was on generality.

First, past research of MES was reviewed, the means to integrate it with other information systems were investigated, and the international standard ISA-95 and B2MML as well as model-driven engineering (MDE) were revised. Next, requirements were defined for the system. The requirements were divided into user and developer requirements. A suitable design for a code generator was introduced and, after that, implemented and experimented. The experiment was conducted by reading production data from the database of MES-like Delfoi Planner and then transforming that data to B2MML-styled XML-schema.

The experiment verified that the code generator functioned as intended. However, compared to a manually created program, the generated code was longer and less efficient. It should also be considered that adopting MDE methods takes time. Therefore, for MDE to be better than traditional programming, the code generator has to be used multiple times in order to achieve the benefits and the systems cannot be too time-critical either. Based on the findings, it can be said, that model-driven application integration methods can be used to integrate MESs, but there are restrictions.

Keywords MDE, software integration, MES, ERP, ISA-95, B2MML

Preface

I would like to thank my instructor Ilkka Seilonen for patiently mentoring me through this thesis. I would also like to thank professor Valeriy Vyatkin for his patience and encouragement.I am grateful to my sister and her husband for the extra check to spelling and grammar. Many thanks also belong to my friends and family for their continuous support along the way.

Otaniemi, 28.5.2018

Tero Takala

Contents

Abbreviations

B2MML	Business to Manufacturing Markup Language
CIM	Computation Independent Model
DBMS	Database Management System
DCS	Distributed Control System
DSML	Domain-specific Modelling Language
EMF	Eclipce Modelling Framework
ERP	Enterprise Resource Planning
GPML	General-purpose Modelling Language
IDE	Integrated Development Environment
ISA	Instrumentation, Systems, and Automation Society
JDBC	Java Database Connectivity
ODBC	Open Database Connectivity
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MES	Manufacturing Execution Systems
MESA	Manufacturing Execution Solutions Association
MOF	Meta Object Facility
MOM	Manufacturing Operations Management
MOFM2T	MOF model-to-text Language
OCL	Object Constraint Language
OMG	Object Management Group
PDM	Product Definition Management
PIM	Platform Independent Model
PLC	Programmable Logic Controller
PLM	Product Lifecycle Management
PSM	Platform Specific Model
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
XML	eXtensible Markup Language
XSD	XML schema
QVT	Query/View/Transformation

1 Introduction

1.1 Background

Different systems, devices and software more often than not have a need to communicate with each other. This interoperability is a crucial part of control systems, industrial automation as well as everyday software people use. Interestingly integrating different systems and software is one of the greatest problems in the software industry [1]. An obvious reason behind the problem is that different developers make different software that have different designs. Thus there might not be any common interface for different softwares to use.

Industrial software development is scattered in to multiple businesses and software is often developed to meet the specific requirements of customers. This leads to customized solutions since different customers have different needs. Industrial systems are also often built incrementally by multiple different developers. One part of the system is made by one contractor while other older part has been made by someone else. Since there are not that many widely accepted common standards to communication this leads to combatibility problems and potentially requires wrappers, adapters, drivers or custom solutions.

Even though integration problems causes a lot of work, standardization of communication is not in everyone's interests either. Different developers want to have support for different things and older systems would not support these new communications without upgrading anyways. There can also be business reasons not to directly support competing system interfaces and there are proprietary drivers as well.

Maintaining software becomes a challenge eventually in a situation where solutions are unique. For example, a customer orders a system and after two years wants to upgrade it. The developer who wrote the code would not remember it after few years or the person might not even work in the company anymore. It can be more manageable to rewrite whole parts of the program instead of trying to guess what the original programmer intended. Documentation does not solve this problem either since it costs and slows development down. Both the customer and the developer have interest to keep costs as low as possible while getting the program working with reasonable amout of work.

One possible solution to the problem is model-driven engineering (MDE). By creating models code can be generated from it. Source code generation has potential to simplify parts of the programming process. Making models is usually assumed to be easier than coding which allows different field specialists to create these models as well [2]. Maintaining software lifecycle is usually expected to become easier as well since by simply updating the model new code can be generated to match latest needs. MDE has a great strength in reusability as well since models can be changed to different models as long as they follow the required structure and thus new code for new application can be

generated. All of these can cut development time and cost significantly and bring advantage when competing in the market.

Integration of manufacturing execution system (MES) to the enterprise resource planning (ERP) is the test case of the thesis but the problem in general is about data transfer between any systems. While still somewhat rare, MES, was chosen to be the test subject since it has been gaining increasing adoption in the industry. From the MDE prespective it makes a good example for data transfer between systems since MDE is not tied to specific systems. Models can be platform independent and they can be replaced with different models and as long as structural rules are followed generating the data transformation code should be possible.

ISA-95 is an international standard for the integration of enterprise and control systems that has potential to gain ground among industrial software developers [3]. The relevancy of this technology to this work is that it can be used to integrate MES into automation and information systems. ISA-95 can provide useful design aid when planning communication between ERP and MES.

1.2 Research objectives

The main objectives of this thesis are to gain knowledge of application integration for MES and usage of MDE-methods in the integration. There are several questions along the way to the these objectives. How to use MDE methods in the integration process? To what extent generalization of the code generation can be achieved and what kind of limitations there are? What are the advantages and the disadvantages of this solution in comparison to manually programmed solution? What kind of technical problems there are to be expected when creating a code generator? Is using this technique worth it and what kind of future work there might be.

1.3 Research methods

The selected research method is an experimentation. MDE methods are evaluated by creating an experiment in the context of MES to ERP data transfer. The setup of the experiment is following: There are two information systems, MES and the other information system. MES has a reading interface and the other information system has a writing interface. Between these interfaces there is a program that does data transformation. The MES used is Delfoi Planner that keeps its data stored in its database. The data is extracted manually from the database and it is stored in temporal variables for the transformation system to use. The other information system is simulated with a ISA-95/B2MML- styled message, that is written with the writing interface. Models used are based on the structure of the data used. These models are then mapped together manually and together they form a combined model. It is known which parts of the models are to be mapped to each other. Code generator, that is made with Acceleo, uses the combined

model data to generate code that does the data transformation between the data structures. Different kinds of tests are created by modifying the models. Only exporting data from MES is experimented on.

The created solution is then compared to a manually programmed solution that does the same data transfer from data extracted from the MES to B2MML structured data. Reading from the database and writing the B2MML document from the B2MML structured data are kept separate from this data transferer and both the manual solution and the created solution use these same interfaces. Lenght of the code, it's efficiency and major differences between solutions are compared. Based on the information gained this thesis tries to evaluate MDE methods suitability for the integration of MES.

Literature research is also conducted in order to gain background knowledge about MES and how to integrate it with other information systems. ISA-95 and B2MML were reviewed and literature research was performed on MDE as well.

1.4 Outline

The structure of this thesis is following: Chapter 2 and 3 focus on the literature study in order to gain backround information of the subject. Chapter 2 contains a literature review on software integration of Manufacturing Execution Systems. It covers an overview of MES and its functionalities as well as its integration with other information systems. This chapter also covers usage of ISA-95 and B2MML when integrating MES. Chapter 3 describes model driven engineering and how it can be used as well as the benefits of MDE based solutions. The focus on this chapter is on the model to text code generation. Chapters 4, 5, 6 describe requirements, design, implementation and testing of the test case. Chapter 7 has conclusions and future work.

2 Software integration of MES

MES are developed to fill the gap between ERP, which is used in business and plant floor control systems such as programmable logic controllers (PLC) and workstations [4]. Ideally the whole system works flawlessly and ERP commands cause independent workstations to operate but in the real world that is not so simple. The origins of MES come from early 1980s data collection systems. Managing manufacturing processes used to be manual labor before digitalization. MES and its predecessors were developed to improve production and reduce its costs.

MES is functionally a system that handles production orders, plans what and when control systems execute, monitors production and relays data to other systems such as ERP, product lifecycle management (PLM) and product data management (PDM). All this is can be achieved faster than it would be possible without a MES [4].

Figure 1 details the hierarchical place of a MES in a system with an ERP and a MES installed. Usage of the system could be following: the ERP gets an order from a customer. Necessary details are delivered to the MES as a production order and then the MES is used to schedule control systems to manufacture what was ordered.

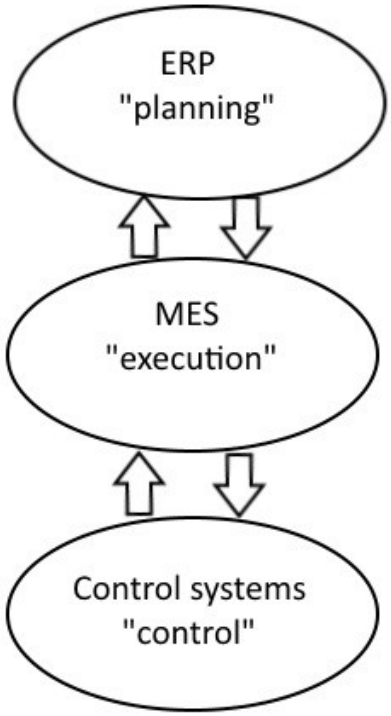

Figure 1: Relation of a MES to an ERP and control systems. Underneath each system name its main responsibility is listed.

There has been few solutions to standardize the way MES operates. Perhaps most notable being ISA-95 that nowadays is managed by Manufacturing Execution Solutions Association (MESA) [4]. ISA-95 divides Manufacturing operations management (MOM)

elements into four types of operations: production, quality, inventory and maintenance operations, illustrated in figure 2 [5]. Production operations management handles production orders and is used to schedule production. Inventory operations management is resposible for availability and storing of resources and products. Quality operations management consists of a group of activities that tracks quality. Maintenance operations management allocate equipment and tools related to the maintaining of the assests to ensure their availability for manufacturing.

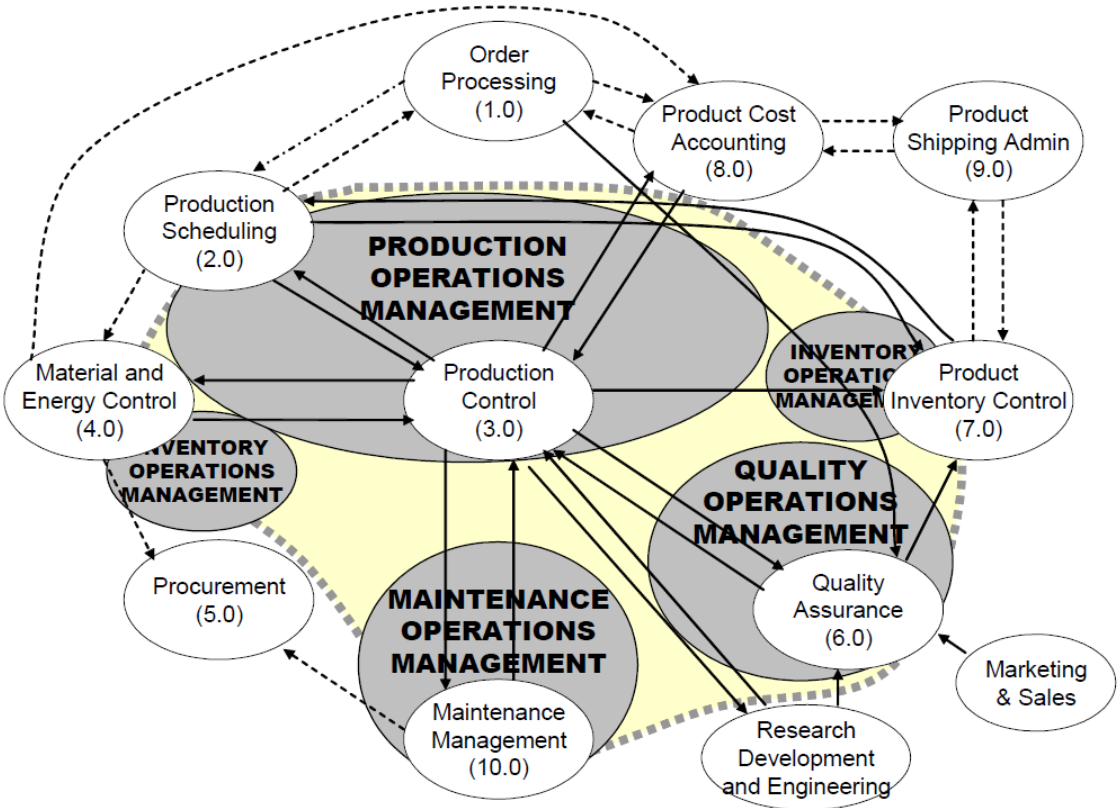

Figure 2: Manufacturing operations management model [5]

Figure 3 represents contents of production operations management from the figure 2. Four "main" lines can be seen in figure 3 in vertical direction. The first line is product definition. The second line represents production capability and activities related to it. The third line is production schedule and the fourth one is production performance. For example, production scheduling needs knowledge of resources and information about production status. Production level 1-2 functions contain access to systems like distributed control systems (DCS). Other three operations can be opened in a similar activity model.

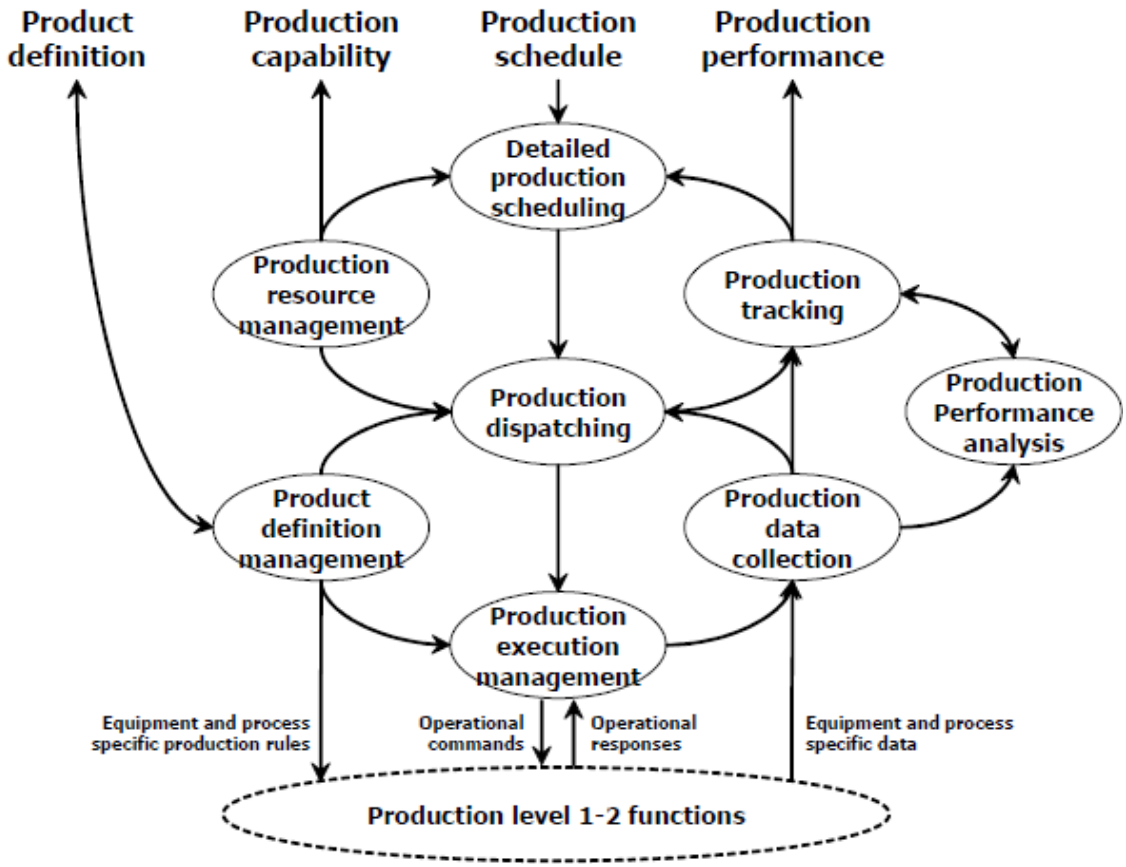

Figure 3: Activity model of production operations management. Lines represent data flows and ellipses represent activities.[5]

There are two architectural variants of MES. The first is application-centered systems and the second is database-centered systems [6]. The application-centered variant has a simpler data structure and allows usage of high-level programming languages, but potential logical errors in system are hard to deal with. Database-centered variant does not allow the use of high-level languages but gains potentially better performance, which is an advantage for a MES. Regardless of the approach, a MES is essentially a database application.

There are adapters, services and business logic around the database that take care of the application and administrational functions that need to be performed. Since MES are used in various environments, the ability to integrate the MES into other systems is important. MES are usually long term investments and the number of systems under an individual MES can range from few workstations to multiple different production facilities[4].

MES are developed to reduce costs at manufacturing, however the advantages of MES are not always clear to a potential customer. This is because MES is an expensive long-term investment and the process of adapting one into your own system is not straightforward. Lack of support between systems requires adapters. In order to MES being able to quickly respond to plant level events, information flow between systems needs to be fast enough. However, old and expensive automated manufacturing systems are often not integrated at workshops [7]. This creates additional amount of work if a MES is to be used and makes the adapting process more expensive.

2.1 Integration of MES with other information systems

The objective of software integration is to connect different systems together to support the exchange of information at service and information levels. Software integration can take place internally or connect different enterprises [8]. Regardless of the method of integration, there is almost always a need to transform data from one representation to another because of semantical differences. This causes a need for rules that define how the integration is done and obligations to route the information to correct destination.

Software integration is often a complex problem. Integration of thousands of applications has not been done, and most projects are integrated at entry level [8]. Nowadays focus of software integration is at service-oriented integration. Software integration reduces costs and makes systems work more fluently. There is a great need for software integration in the future because of it providing an enterprise with an infrastructure that can handle business electronically and in real time.

One typical connection between an ERP and a MES is that the ERP delivers production orders to the MES and the MES returns data of what has been done to the ERP. Sometimes, additional data is needed, but usually the ERP does not need to have a large amount of information about the processes ongoing in the MES. When including a MES into a system that already has an ERP, it is often found that interfaces do not match without additional adapters and wrappers.

Integrating multiple databases into one is difficult and a database management system (DBMS) is not well suited for the task [9]. Hence other methods for integration are needed. One way to integrate a MES is to utilize Java Database Connectivity (JDBC) or Open Database Connectivity(ODBC). By accessing databases directly, data can be written and gathered by using queries. Gathered data can, for example, then be formatted into XML.

Schemas and the role of XML cannot be ignored when it comes to data integration over recent years. XML provided much needed common format for syntax [1]. XML is used in various business processes to share data, and many webservices utilize XML.

Webservices are a technique used to integrate software over a network [10]. Services provide interfaces that other software can use to access the system. Interaction with webservices is often done using simple object access protocol (SOAP) where XML messages are conveyed using HTTP. XML provides a flexible platform for messaging and makes webservices a dynamic solution.

2.1.1 ISA-95

ISA-95 standard can be used for enterprise integration as well [11]. Chapter 2 stated that ISA-95 significantly contributes to the functions of MES. Certain parts of ISA-95 focus on integration of MES as well. ISA-95 part 3 defines the hierarchy and activities of different levels seen in figure 4. Level 4 represents operations made by an ERP and level 3 the MES domain. Levels 0-2 are production systems.

ISA-95 provides a common terminology and object models that can be used when integrating different software systems. ISA-95 Part 1 and 2 focus on the interfaces between level 3 and 4 [3].

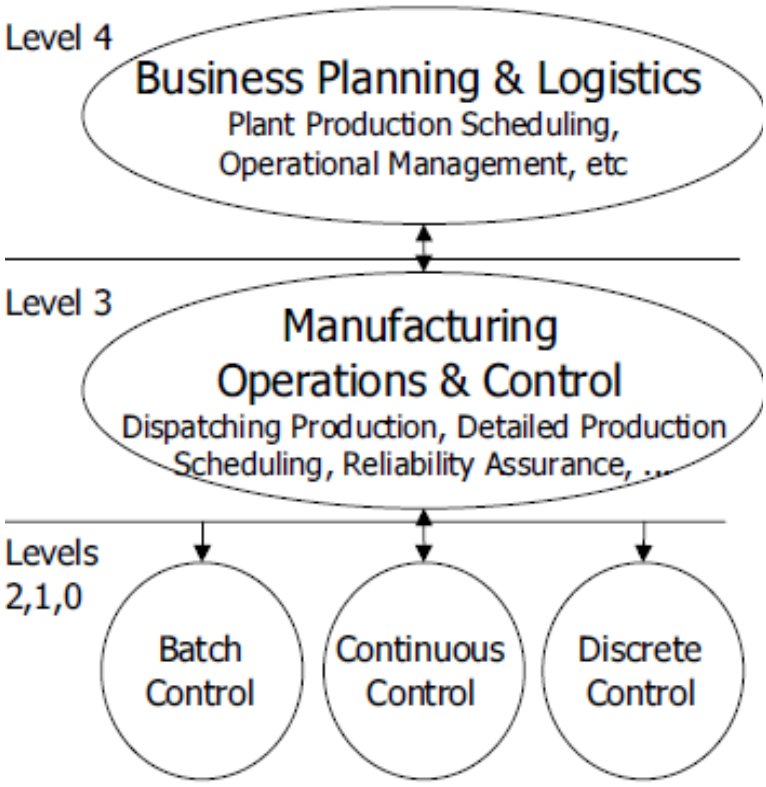

Figure 4: Functional hierarchy of ISA-95. [3]

ISA-95 Part 4 defines the the interfaces between the ERP and the MES with a goal to ease the integration. The scope of the part 4 is limited to the definition of objects models and attributes that are used for the information integration [11]. Attributes and object models have standardised names and descriptions, creating a common terminology. Part 4 also provides information of how object models are related to each other.

2.1.2 B2MML

Common data definition B2MML defined by the ISA-95 standard links MES to level 4 systems. According to MESA, any company can use B2MML without royalties, as long as they give credit to MESA. The latest version is V0600 which brings support for ISA-95 Part 4.[12]

B2MML is essentially an XML implementation of the ISA-95 standard. Its XML schema (XSD) closely follows the data models suggested in the ISA-95 standard. The elements defined range from personnel information to production details. B2MML is intended to be used to link ERP and supply chain management systems to MES.

3 Model-driven engineering

Model-driven engineering (MDE) is a development approach, where models are in the center of the development process. MDE belongs to a hierarchy group known as MD* acronyms. Figure 5 shows four of these acronyms and their mutual relations but other MD* acronyms exist as well [13]. While there are differences between what exactly each name represents, they all share a common goal: to make software development easier and more automated.

Models have a long history in software development. Generally they have been used to help with software design, documentation and visualization. In MDE, models are seen as equivalent to code and are raised from mere blueprints to integral part of the programming process [14].

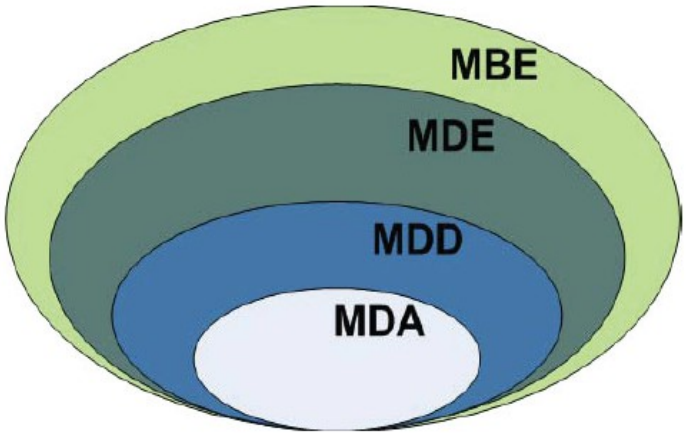

Figure 5:MD hierarchy. Model-based engineering (MBE), model-driven engineering (MDE), Model-driven development (MDD) and model-driven architecture (MDA) are supersets of each other. [13]*

Model-driven architecture (MDA) is developed by Object Management group (OMG), which is a non-profit technology standards consortium. Basic functionality of MDA takes place as follows: first, a platform-independent system and its functions are defined. Next, the system model is transformed for the desired platform. There are different models to support the process. Computation independent model (CIM) contains system requirements, Platform Independent Model (PIM) describes system design and Platform Specific Model (PSM) portrays a system design that is platform-dependent.[15]

A key concept of MDE is transforming models into source code or other models. First, models need to be created. Depending on the software that is developed, the number of models can be different. The advantage of splitting larger models into smaller ones is increased modularity, but just changing parts of the larger model can be viable. Next, the created model can be transformed to a new model or source code can be generated.

3.1 Models and metamodels

Models can be represented with text or with graphics. Many older models before popularity of the UML were text-based. However, with the increasing availability of modelling tools, graphical models have gained popularity. The old saying ” a picture tells more than a thousand words” is a good way to describe models. It can be assumed that models can be easier to understand than code.

Models can be divided into models and metamodels. Metamodels describe the abstract syntax of the lower level models. Relationships, modelling rules and constructs of a modelling language are defined but not the concrete syntax of the language [16]. Metamodels and models have a class-instance relationship shown in figure 6: higher levels describe lower level models and lower level models are instances of higher level models and conform to them.

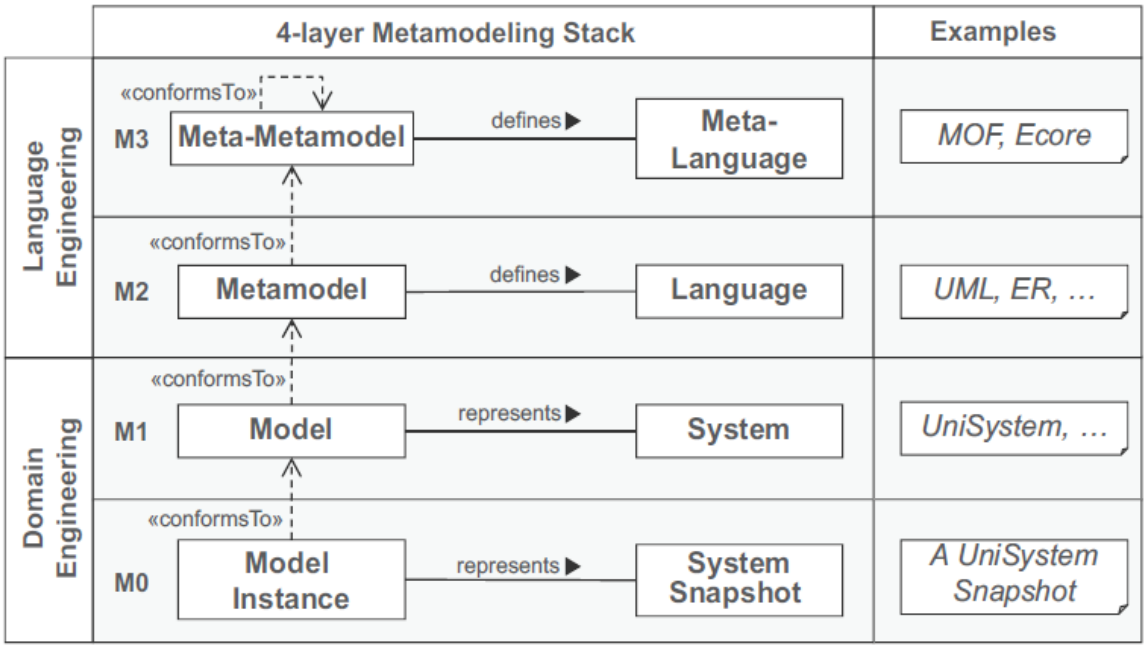

Figure 6:The four layers of metamodels. [13]

Models represent the system and a model instance represents a snapshot of a system. These are domain engineering layers. A metamodel defines a language and a meta-metamodel defines a meta-language. These make up language engineering layers. This can be seen in figure 6. For example M0 is a snapshot of the system, M1 is model of the system, M2 is then the modeling language used to create that model e.g. UML, and M3 is something that is used to specify the modeling language e.g. Meta object facility (MOF).

Modelling languages can be categorised in to domain-specific modelling languages (DSML) and general-purpose modelling languages (GPML). DSML are languages that are used for specific domains and purposes. GPML are made to be applied to any domain. UML has gained a lot of popularity lately. UML can be seen as GPML, even though it was

developed mainly for modelling software systems [17]. UML provides a visual modelling tool to aid with software development, but it can be used to model business and similar processes as well. UML conforms to MOF-based metamodel that specifies the abstact syntax of the UML. In addition UML has a detailed explanation of the semantics of each UML modeling concept.

MOF defines itself and is used to model other modelling languages such as UML [18]. Other prominent metamodeling language, Eclipse Modeling Frameworks (EMF) language Ecore can also be used to model UML. Ecore is tied to Java implementations, whereas MOF is not.[13]

Object Constraint Language 2.3 (OCL) was developed in parallel with UML 2.0 and MOF 2.0 [19]. It is used to describe expressions on UML models and to define model constraints. OCL is declarative, typed, and free of side effects. Being declarative means that OCL cannot be used imperatively. OCL being a typed language means that every OCL expression has a type and must conform to the type conformance rules of the language. For example, UML model classifiers have OCL types. Freedom of side effects comes from the fact that OCL is a pure specification language. While OCL expressions can return values, it cannot be used to change anything in the model.

3.2 Transformations

Model transformations can be divided to model-to-text, text-to-model, that can be grouped to projections, and to actual model-to-model transformations [13]. Model transformations can be used atomically or chained freely. Figure 7 gives an overview of interoperability between two systems. On M1 layer, input files conform to formats on M2 layer. In the middle there are models that are used in the transformations. These models conform to their corresponding metamodels and the metamodels conform to the meta-metamodels.

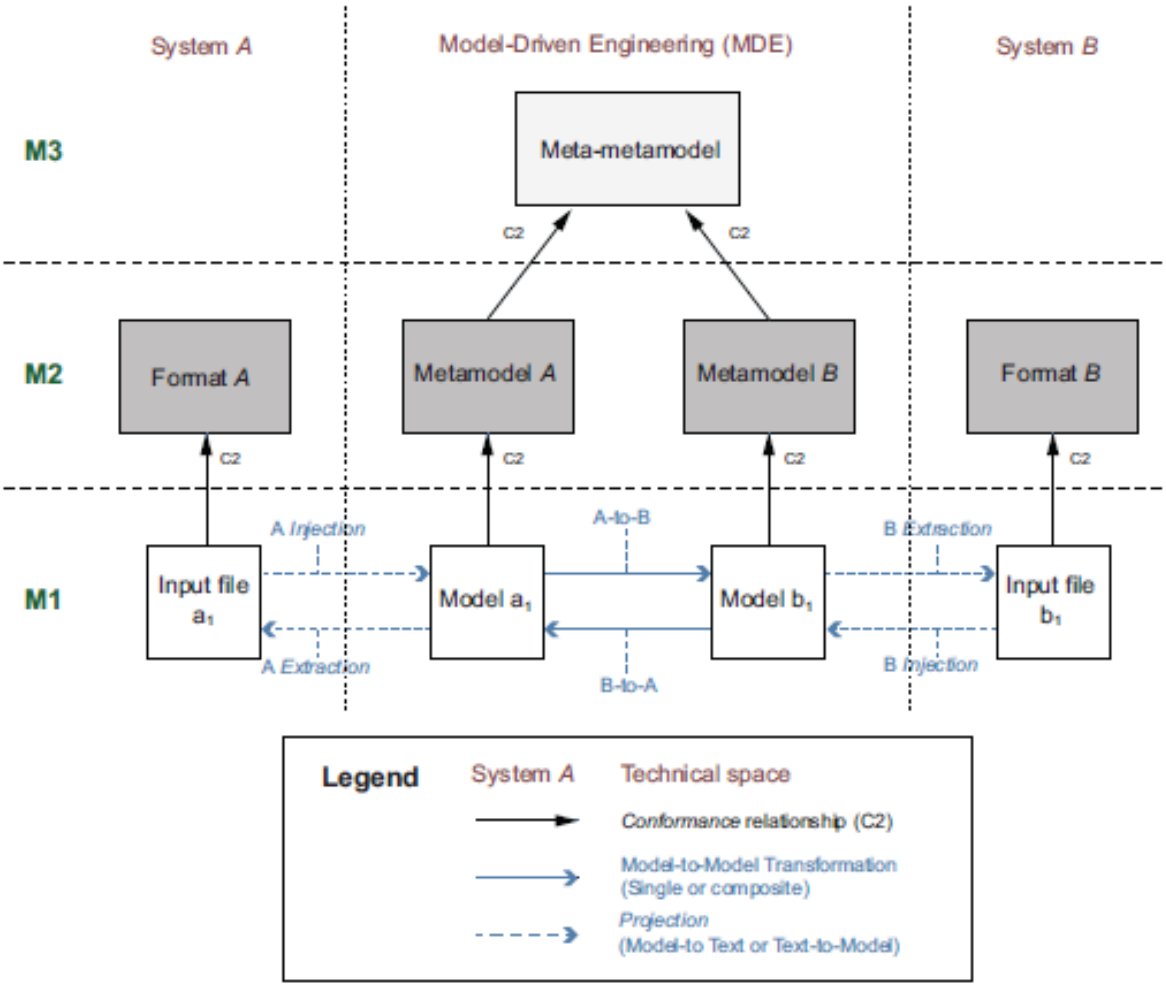

Figure 7: Generic interoperability between two systems. Edited [13]

Text-to-model, i.e. injection, creates models from text. XML-file is a good example for injection since it is structured text. Schema can be read easily to create a model. Depending on the code used to read the XML-file, different things can be achieved. The full file can be read into a model, which can be useful when creating documents about systems, for example, a blueprint of a machine. Creating a model from a schema and metadata can be used to create a template, which represents the XML-file, that can be used in model-to-model transformations.

Model-to-text, i.e. extraction, is the opposite of injection. Reading from a model into text can recreate the XML-file that was used as example in injection. This can also be used to create simple code templates from models to help with coding. The most important function of model-to-text transformations for this thesis is source code generation.

Model-to-model is the actual transformation. One-to-one transformation is often sufficient, especially since transformations can be chained, but transformations can be many-to-one , one-to-many and many-to-many as well.

By definition, mapping links corresponding elements between models. It can be divided to the actual mapping of elements of the model to each other and to automating the generation of transformation rules for the system that has input of two models and the mapping between them.

3.3 Source code generation

Code-generation produces code from a model in order to create programs [13]. This can range from the whole program to certain features to class initiation. Code generators can also be flexibly expanded. One way to generate code is to use rule-based templates as code generators. The templates have placeholders where data gathered from the models is input according to preset rules in the template and the code is created.

```
[template public find_Glb_class(p:Property, setValue:Sequence(OclAny), print:Integer,
          listname:String, part1name:String, parttype:Sequence(OclAny))]
    [for (c:Class |p.ancestors(Package).eContents(Class))]
        [if p.type.name = c.name]
            [if print = 1]
                //TEST!!
                [listname/] [part1name/][c.name/]_variable = new [listname/]();

                [part1name/][c.name/].add([part1name/][c.name/]_variable);
                [part1name/][c.name/]_variable.[parttype/] = [c.attribute.name/];
            [/if]
            [if print = 0]
                [setValue/][c.attribute.name/];
            [/if]
        [/if]
    [/for]
[/template]
```

Figure 8: Example of a template made with Acceleo.

In figure 8, the text written in red surrounds the functionality of the template containing the name and parameters of the template. This template is public and its called find_Glb_class. It searches all of the ancestors from the package that are classes and cycles through them. The print parameter decides what kind of information is generated. The square bracketed parameters generate the said parameter to the code. Text without brackets is generated as it is. Information from variable c is generated using dot notation where [c.name/] generates name of the class and [c.attribute.name/] generates what is written in the attribute name. This template does not call a new template but it could do so simply by writing name of the desired template and giving it proper parameters.

Once the code is created it can be tweaked freely like any other code. Features can be added and modifications can be made. This can be convenient since certain features can be much harder to generate from models than manually coded. One thing to remember is that changes that are not done at the model or generator level will be overwritten unless they are protected. Naturally, changes will not carry over if the models or the generator are reused somewhere else. Parts of the code can be protected so that a new generation cycle

will not overwrite the defined protected parts. This can be used to protect manual changes and added features.

There are other ways for code generation than just templating. To name a few techniques, there are templates + filter, templates + metamodel and api-generators. There are also multiple tools other than Acceleo available [20]. Acceleo uses template files, which are used to define sets of rules for converting models to source code.

3.4 Tools

Eclipse is a commonly known open source integrated development environment (IDE). It has a broad plugin support to meet the needs of developers. Several tools suitable for MDE are avaivable as plugins. Eclipse plugins contain multiple modelling plugins and some code generator plugins. EMF is a common standard for many frameworks and technologies and many tools in eclipse are based on it [21].

For model-to-model tool support, Eclipse has EMF based projects, such as EMF tiger, Henshin, Fujaba, e-Motions, ATL Refining. There are multiple templated-based transformation languages that can generate text from models, such as XSLT, JET, Xpand, MOFScript and Acceleo to name a few.

Acceleo is a practical implementation of the OMGs MOF model-to-text language (MOFM2T) [22]. Its objective is to provide the best tooling possible to generate code. Acceleo is a ongoing project with planned features that have not yet been implemented [20]. Another MOF based mapping language would be QVT [23].

UML is one of the most important tools in model creation. Different UML tools can be used to create suitable models for the MDE process. The UML tool called UML designer provides support for MDE and UML designer is direcly compatible with the templating tool Acceleo [24].

3.5 Adaptation and advantages

One assumed benefit of MDE is that it has potential to speed up the software development process. Reasons for the accelerated development speed are multiple. With one or more transformations, working code can be generated. Similar structured code can be generated fast by reusing generators and/or models.

By making the changes to the generator or to the model, changes can be applied to the programs created with that model or generator. Multiple pieces of code can be modified, updated and fixed by a few clicks of a mouse. Developers no longer have to track down

pieces of code one by one since fixes can be made in on place. Similarly, if there is a bug in the generator the same bug will spread to all pieces of code that have been generated with it. While fixing on the model level can be more challenging, this will eventually raise quality of the code and purge bugs out of it. In the long run, generated code can have better quality than manually written one since humans are prone to mistakes. On the other hand, certain features can be difficult to model and generate and have to be written manually. Extra care is required when updating generated parts so that manually written parts are not overwritten. One way to avoid this is to use protected areas in code generator to protect the manual code.

The MDE approach helps with maintaining the software. Updating models after months or even years is easier than reading thousands of lines of code and trying to remember how exactly did it work. The person who wrote the code might not remember or, in the worst case, does not work at the developing company anymore.

Reusing code is a common practice in software development, but MDE takes this concept one step further. Models are on a higher level than code when it comes to abstraction. By replacing a model with another and using same transformation rules can generate working but different code. Similarly, old models can be recycled and paired with new transformation rules to generate something new.

Often models are used to initialize classes, produce xml, or form simple functions instead of providing more complex solutions. Reasons behind this could be that the initial investment in MDE design is too big, and just by reducing a tedious task like creating classes is enough to speed up the development. While this can be true, the main benefits of MDE approach are not gained in this way.

One of the criticisms againts MDE is that models are more useful for people who cannot program. But, since models have to be complex to make working programs, those who cannot code cannot understand the model either, and those who can code would rather just write the program instead of dealing with the models [13]. While this is true, long term benefits can be large enough to invest in MDE. This is especially true in larger projects since MDE can make tedious and repetitive tasks manageable, for example, models should be good for different kind of report generation and generating initial classes with functions.

Another valid critique is that making models is more time consuming and expensive than just programming the code directly [13]. A partial reason behind this is that it takes time for developers to adjust to the new methods and modelling, and generating certain features can be troublesome. Other reason is that there are no reusable models and generators avaivable for the first adaptation, and everything has to be created from scratch. At the beginning of the adaptation process, MDE is bound to be slower like any new technology in a similar situation.

4 Requirements

4.1 System definition

The objective of the system designed in this thesis is to create a mapper that transfer data in form of business documents from a MES to another IT-system. The principle is demonstrated in Figure 9. A key part of this system is the code generator that uses models from both the MES and the IT-system to create the mapper. The mapper generation is based on information learned from MDE principles[13]. The interfaces allow easier reading and writing of the data.

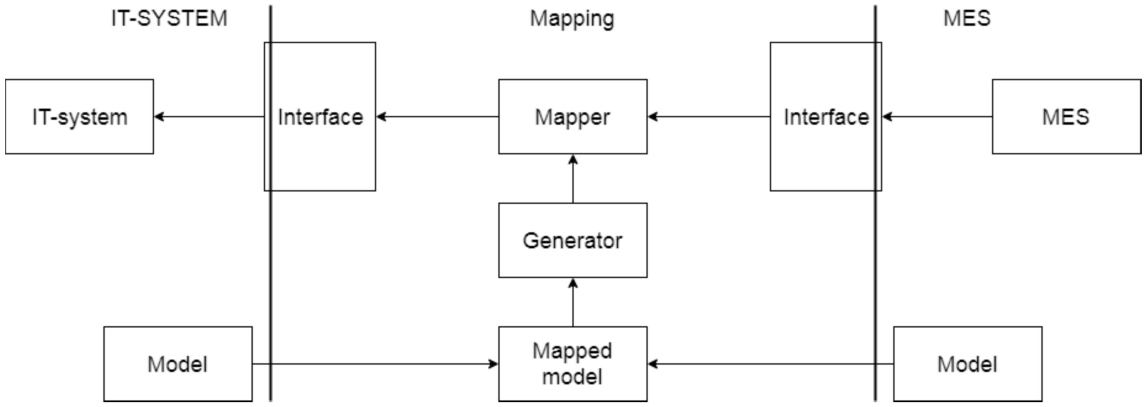

Figure 9: Mapper connects the MES and IT-system through interfaces. Vertical lines split IT-system, MES, and Mapping.

The motivation behind this system is to ease data transfer between a MES and other IT systems. This particular system provides one-way transfer of information from the MES to the IT-system, although many systems need to transfer data on both directions. This system is kept simple in order to demonstrate model-driven application integration for MES. The mapper is traditionally developed manually, but in this thesis, it is generated by MDE methods. Since the mapper is the most complex part to create, the use of MDE for this task can be beneficial.

4.2 User requirements

The user requires a system that transfers data from a MES to another system. Therefore, most of the user requirements are related to the functions the system performs with some additional features. In addition, the ability to upgrade the system in the future is in the interest of the user, as industrial software is a long term investment and often new software are added to it during its life-cycle. Related to this, the system needs to be maintainable.

As a consequence of the data transfer requirement, what is required is an operation that collects data from one system and transforms it for another system to use. The structure of the part of the MES SQL database used needs to be known and it cannot change during use. First, data is read from the MES SQL database through an interface, whose structure is compatible to the tables of the database. Depending on the data needed, the universality of the reading interface can vary. Naturally, reading less information is faster. A Model that is used in code generation represents the structure of the data that is read. Similarly, on the target system side, the desired data structure is represented by a model. The writing interface is selected to output the desired format. In this case, the user wants to read a certain production order from the MES and write it into B2MML -formatted xml-file. The mapper, which is in the middle of the system in figure 9, does the data transformation and is generated from combined models.

4.3 Designer requirements

From a designer's perspective, financial concerns are usually the driving force behind all decision-making. Industrial software is often customized case by case, and new solutions are created instead of using already made programs. The designer wants a solution that saves workhours and requires less programming to create. Because of this, the mapping of the data has to be done with models. And the code that performs the mapping has to be generated.

During the software lifecycle, the software is usually updated to support new features. The user might want to integrate new software in the future or export of new kinds of business documents from the system. Therefore, the developer, who designs the software and has a contract to maintain it, has an interest to keep maintenance work easy. System updates can have a long time intervals. During this time, the person who wrote the code may forget how the code worked and the documentation (or lack of it) will only help to a degree. This can lead to situations where it is easier to create new a solution instead of updating the old one. Maintainability is a important designer requirement since maintaining software is usually the largest part of software life-cycle costs [25].

Reusing already functioning solutions is a common aspect of software development. The problem with reuse is that industrial software often requires customization. A custom solution is harder to reuse for other projects as well. Reusability of the generator is required from the system.

In industrial software development, specialists of different fields often work together. While many technological specialists have some background in programming, there are also a lot of specialists that do not. Therefore the specialists that cannot understand much of the code have a harder time to participate in creation of the system. Anything that can make it easier to use specialists that might not have strong programming backround in the project is a clear advantage.

Applying the MDE approach properly can help to cover aforementioned designer requirements. The MDE approach is in a key position in this thesis, and with it, a general solution to similar problems becomes a possibility. Maintaining software should become easier with the MDE approach. By replacing and updating models, working code can be generated for new features as long as they are not too different, i.e. the structure of the data is not in a format that cannot be converted into a tree structure. This is especially useful when generating documents of different content or format. MDE-methods have also taken reusing in to account, and parts of the code generator and models can be applied to other MDE projects. Code has to be tested before deploying, but with code generation, testing and creation of potential fixes should be easier. It is also assumed that models are easier to understand than source code. This can also benefit specialists that work in the project that have limited knowledge about coding.

Furthermore, there are the following requirements regarding the models and the generator. The way the data is structured and how models are mapped into each other are relevant information for the designer of the code generator since the generator and models need to follow the same naming rules. This means that it is known what each name means and represents. An example of this is that same names are used in code generator and models. For example, Id is known to mean Id in both the model and the generator. In addition, the models used must to be tree shaped or they have to be simplified into tree shape.

5 Design

5.1 System architecture

The architecture for integrating the MES into IT-system is shown in figure 10. It consists of two IT-systems that are connected through a mapper. The mapper is to be generated from models by using a code generator. The models are based on the class tree structures that are used to temporarily store the information gathered.

First, data is read from an IT-system on the right of the figure 10. That data is stored in a database from which it can be accessed either by using premade functions or by directly accessing the database. The interface is used to access the IT-system in order to avoid having to write complicated reading functions in the mapping part of the system. Similarly, on the writing side of the system, there is an interface to keep the possibly complicated writing operations separate from the mapper. Writing takes place with the help of a suitable API from the other interface.

The mapper is a program that describes how one data structure can be transformed to another. It contains functions to retrieve data from the interfaces and functions to manipulate and reorganise the data. The mapper is generated by using code generation. In order to generate the mapper, suitable models are created for the code generator to use. These models are derived from the interfaces and can either be manually created or generated from the interfaces. These models are then mapped to each other so that the generator can use this information and gain information of external details in the mapping if there is any. The code generator is then used to generate the mapper-code.

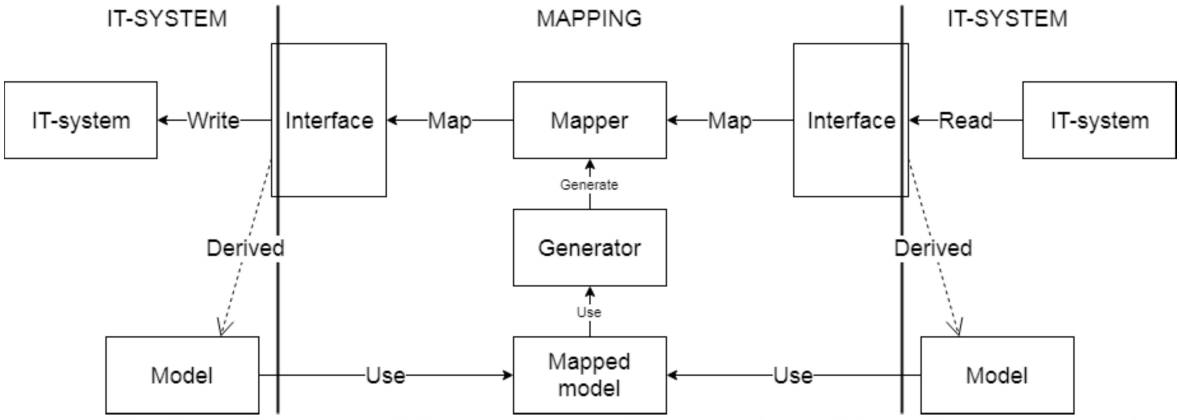

Figure 10: The architecture of the system. Interfaces are derived from their corresponding systems and models are derived from interfaces. Arrows indicate the direction of data flow.

5.2 Modules

5.2.1 Interfaces

Reading and writing interfaces represent their corresponding systems and the way the data is stored inside class-tree structures. Through these interfaces, writing and reading is done between corresponding IT-systems and the class-tree structures, which are used to temporarily store the information. The API can deal with the used datatypes and it can be described as UML model.

5.2.2 UML Models

The models used are UML class diagrams. There can be more than just one model to represent the IT-systems. Regardless of the number of the models, they are combined through mapping to allow the code generator to determine how the data from one model is connected into the other one. The generator and models follow the same naming rules. At least three UML models are needed: the input model, the output model, and the mapped combination of the two.

5.2.3 Code generator

The code generator uses the combined UML model that defines the mapping and a code generation technique to create the mapper. Here, the code generation is done by templating. The mapping of the models for the combined model is done manually by the developer. The code generator is created, or the existing code generator is configured for the current configuration of the system before usage. Navigating through the mapped model and formatting the data into proper style is made with the code generator. The designed generator can handle a combined model that has multiple input UML models and one output model (since the algorithm starts from the top node of the output model). If multiple output models are desired in the future, the generator can simply be reused and proper output model can be added to the combined model. UML models are modified into the naming rules of the generator. As presented in section 4, proper naming and tree structure for the models are required. If the UML model cannot be modified into tree structure, it will not work with the designed generator.

5.2.4 Mapper

The mapper transfers data from one interface to another interface. These interfaces handle different data stuctures and need to be accessed in order to transform the data from one structure into another.

5.3 Functions

5.3.1 Configuration time functionality

The objective of the configuration time is to generate a working mapper through code generation which uses the combined UML model to generate the code. It is known what kind of data is interesting when extracting the data and thus the interface can select to read only the data wanted. Since the UML models represent the data structure they are also assumed to be known at this point. Following steps must be taken in order to reach this goal.

The first step is to map the UML models together to allow the code generator to generate a working program from them. This mapping is done manually by the developer. How the mapping of the models is done is based on the data and the knowledge of attribute representation with respect to the other model. Naming rules are followed so that the code generator is compatible with the models. The code generator navigates the combined models and handles the data. On the figure 11 inside the receiving model there are many classes and many of those classes have attributes. Similarly, the models on the right have classes and the classes have attributes. The attributes have associations to each other. One attribute can have multiple associations. These associations are the mapping that has been done to combine the models into one big model for the code generator to process. Mapping of the attributes is based on the data structures used.

Figure 11: Simplification of the models mapped to eachother. On the left is the receiving model and on the right is the combination of the model from which data will be extracted.

Figure 12 shows a part of the receiving model. Nodes on the left are classes that make up the tree shape of the receiving model. The classes contain multiple helper attributes for the code generator. For example, *hax_up_list* contains the name of the association that leads towards the top in the tree. Names of these helper attributes are a part of the naming rules that are followed so that the code generator and the models can be used together. Association *has_attribute* leads from a class to an attribute class. In order to use the models properly, attributes have been given their own nodes. The reason for this is that it was not found possible to select one attribute among multiple attributes and keep the system general enough thus keeping the benefits that come with the MDE. These attribute classes have names that start with Atr and actual attributes have their names inside them. These

attributes are mapped to the other model and one attribute can consist of multiple attibutes. Finally, there are global values that start with Glb. The purpose of these values is that there can be data, for example IDs, that are added to the new data structure and are not available in the old one.

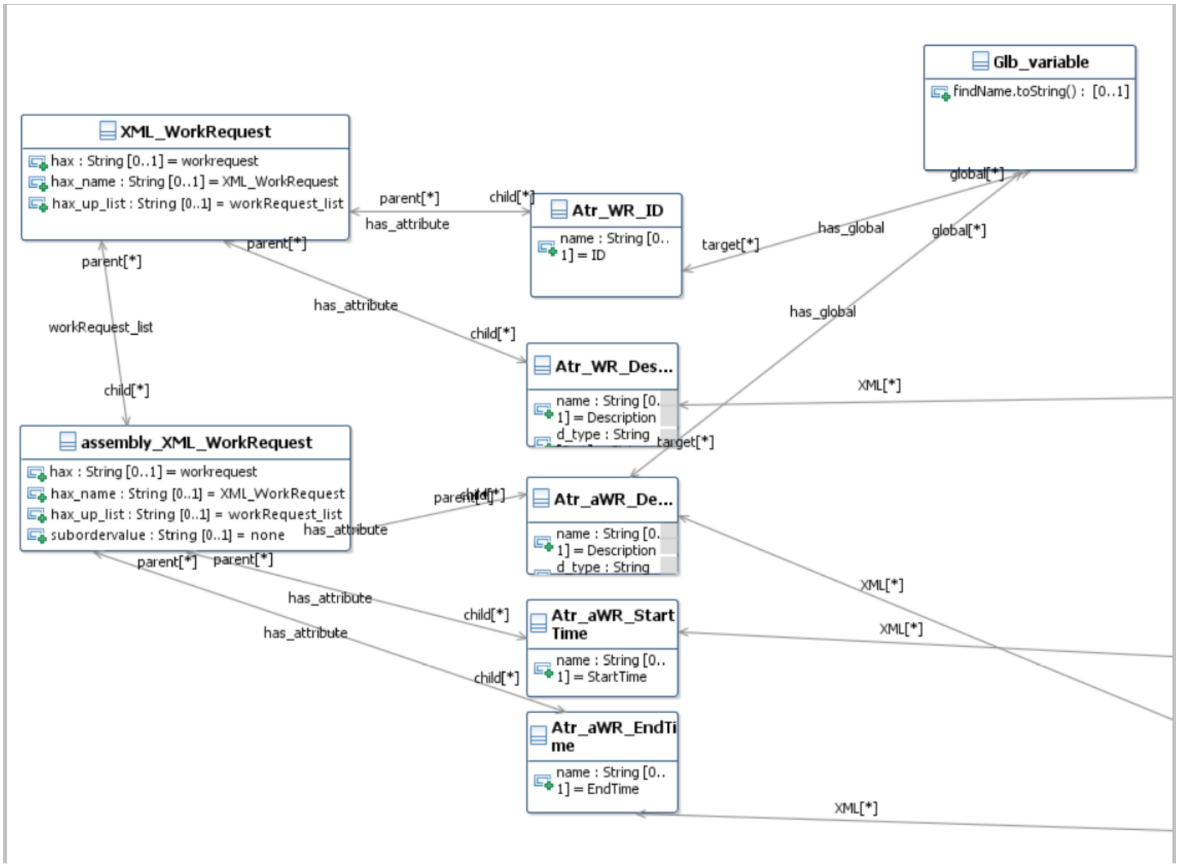

Figure 12: A simplified portion of the recieving model and its associations.

The next step is the configuration of the code generator. Used models are set as active models for the generator and the starting point is set. The code generator scans through the model systematically and creates the required data manipulation inside the mapper. After all the configuration is done, the generator generates a functioning mapper with a press of a button.

5.3.2 Functionality of the code generator

The logic of the code generator and the generated mapper is the following. The generator algorithm gathers the attributes and names of the classes from the UML models. The generator contains functions for navigating the model, gathering data, writing code, and adding data from the models to the code that is to be generated. Then it writes the mapper that operates with the interfaces and the actual data. There are two tree-shaped models in the combined model source and target. The algorithm governing the designed code generator works with the tree-based data structures and it can be split into three main parts: Selecting the path down the target tree, collection of the mapped attributes, and tracking

the path up the source tree. The algorithm will start from the top node of the target tree. While it descends down the target tree it will fetch the path to data from the other tree by climbing it up.

Each parent node that has attributes triggers the mapped attributes section for every attribute of the parent node and each of those mapped attribute sections will likewise trigger the last algorithm part that tracks the path up the other tree shaped model.

Selecting the path down the target tree (figure 13) : The code generator algorithm starts from the top node of the target tree and proceeds down from the top. First the algorithm checks for attributes. If attributes are found, it proceeds to the attribute and initiates, the collection of the mapped attributes. This is done for every attribute linked to the node. These attributes are gathered together under an instance of the class. Then the algorithm checks for the child nodes of the current class node. Once a child node is entered, the subtree under it is processed with the same logic. This is repeated until the whole tree has been processed.

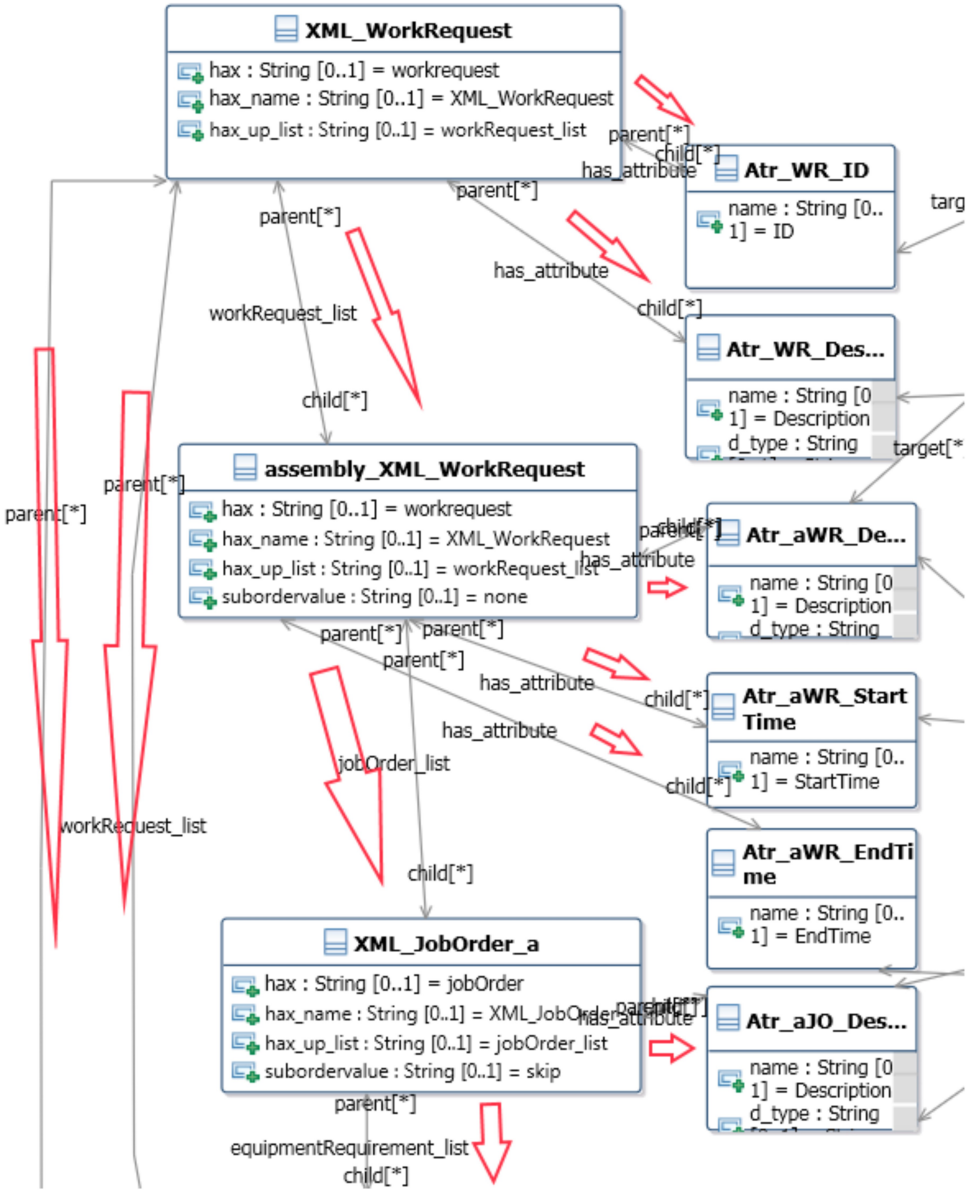

Figure 13: Algorithm proceeds systematically down from the top. Red arrows indicate the direction of the algorithm. Nodes that start with 'Atr' are attributes.

Collection of the mapped attributes (figure 14): The attributes of the target model are mapped into the attributes of the source model or into the global variables. Target attributes can consists of multiple attributes and each of these mappings are followed. If the other end is a part of the source tree, the algorithm proceeds to ”tracking the path up the source tree” section. The reason behind this is that a proper path to the attribute must be found from the other tree in order to generate the code that fetches the values. Global variables do not have parent nodes to track, therefore there is nothing else for the algorithm to do other than read the data. Once all mappings have been processed, the algorithm returns to earlier section.

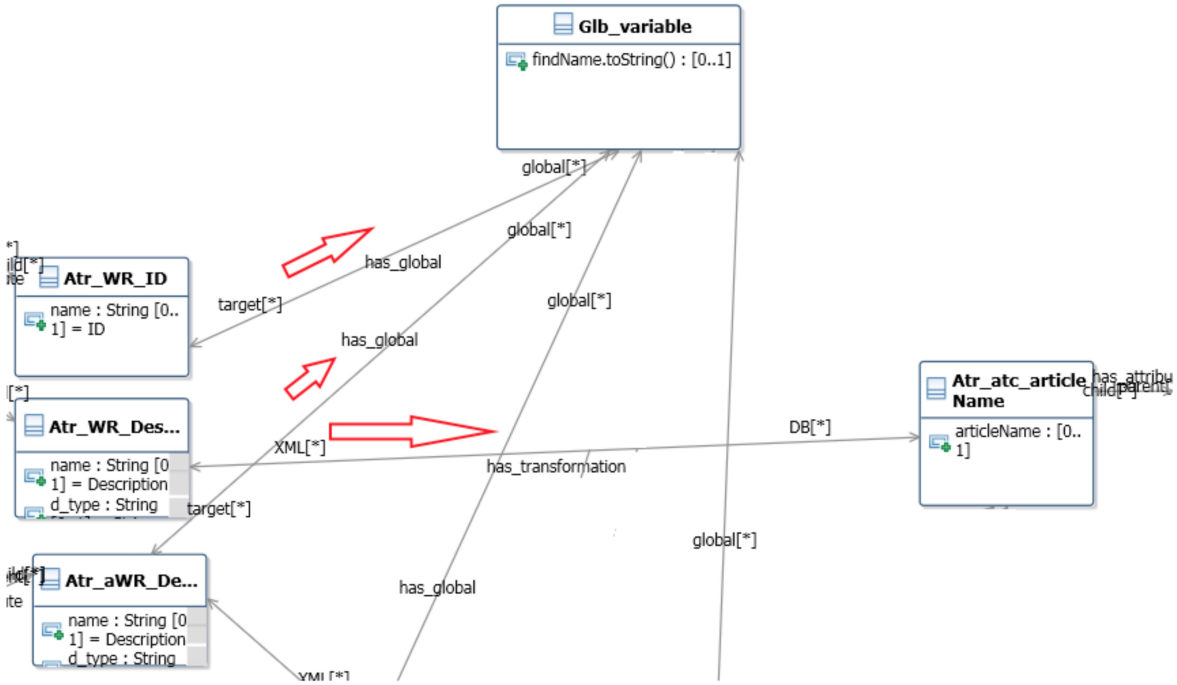

Figure 14: Algorithm finds corresponding attributes and their location in the model. Red arrows indicate the direction of the algorithm. 'Glb' is global variable and is not connected to a tree. In contrast, all attributes are connected to a tree.

Tracking the path up the source tree (figure 15): From the attribute the algorithm starts to find its way to the top node of the source tree. Route to the top is recorded so that the code can be written. By using helper attributes, the algorithm proceeds one node at the time towards the top. Once the top is reached, the algorithm returns to earlier sections.

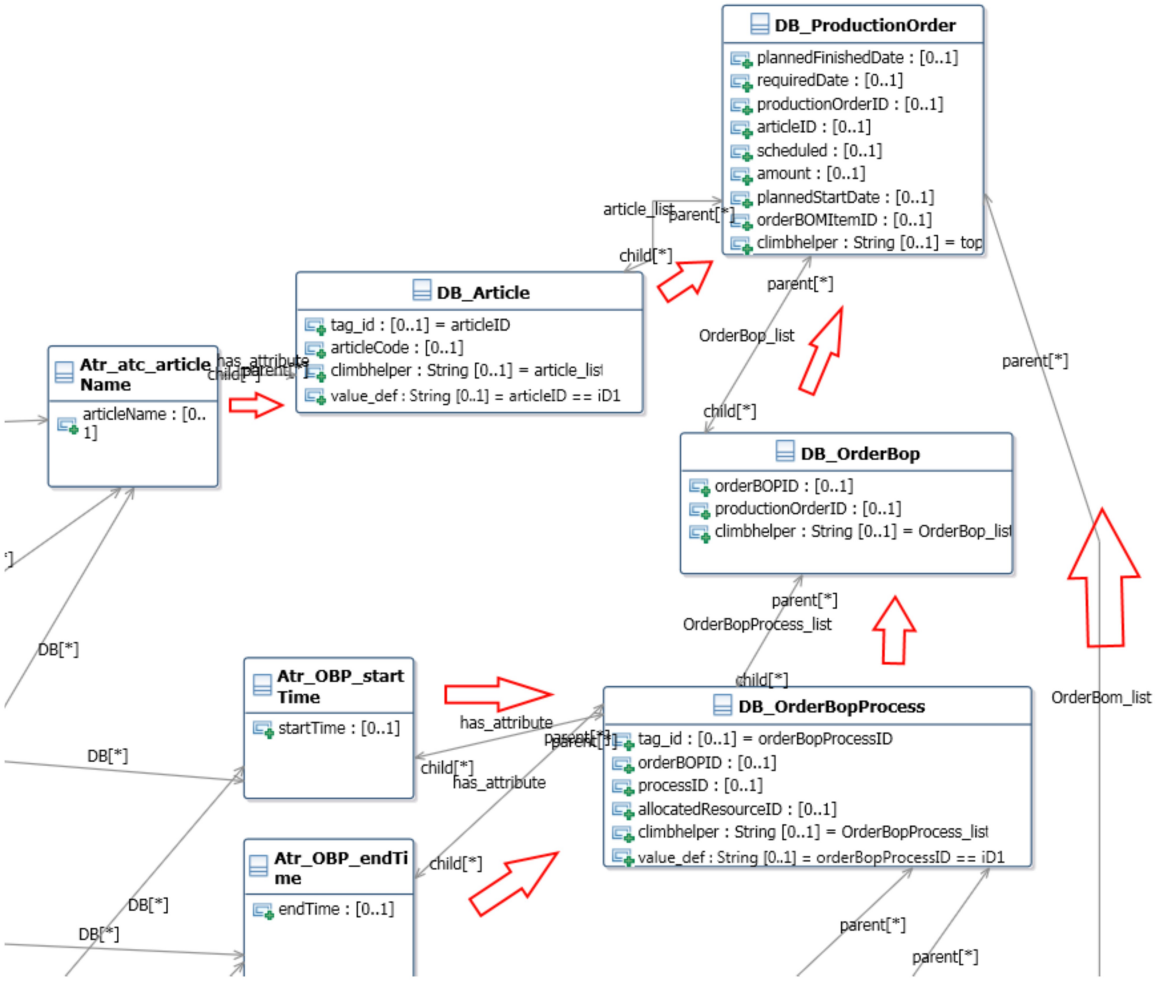

Figure 15: Algorithm proceeds towards the topnode of the tree recording the path used. Red arrows indicate the direction of the algorithm. Nodes that start with 'Atr' are attributes. DB_ProductionOrder is the top node of the tree.

5.3.3 Runtime functionality

After the configuration of the code generator and the generation of the mapper, the system can be run. Runtime functionality is limited to reading, transforming, and writing functions.

When the system is started, reading data from the IT-system database (MES) is executed. API is used to access the database and the required information is stored into the class-tree structure of the interface. Once reading is done, the mapper function transforms the data from one structure into another and the writing interface receives the transformed structure. The writing interface then proceeds to write the data into the desired format. After that, the data is delivered to the next IT-system.

6 Implementation and experimentation

6.1 Implementation

The implementation is a prototype derived from the design presented in Chapter 5 (Figure 16). Code generator is created for an UML model that consists of two tree-like models that have been mapped to each other. The IT-system from which data is extracted is a MES-like Delfoi planner. Delfoi planner stores its data into a SQL database. The information is retrieved straight from the SQL database since there was no usable API available for the Delfoi planner at the time of the implementation. Java was used to create the prototype, leading to use of JDBC (see ch 2.1) for the data extraction. Test project data is found and copied with JDBC. Navigation inside the database tables is done with SQL commands and key-id values.

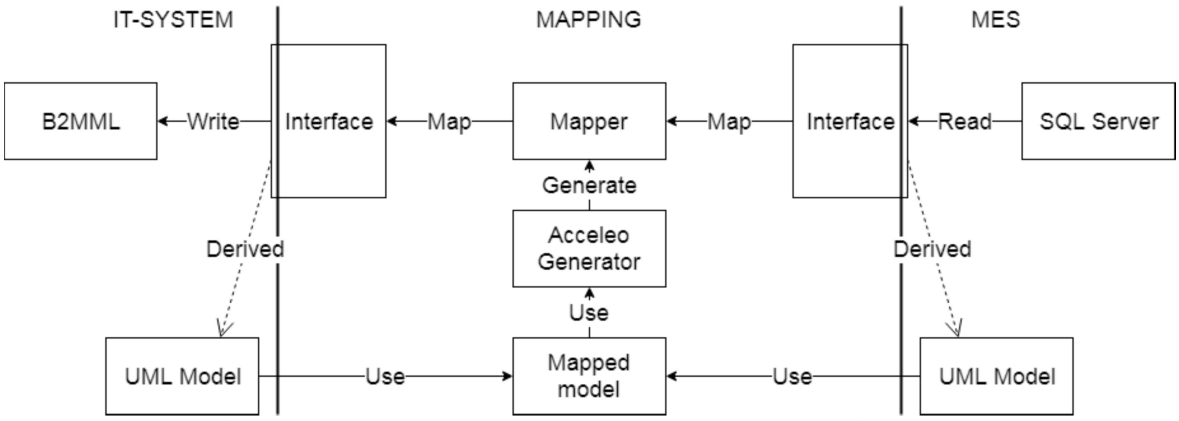

Figure 16: The development environment of the prototype. Acceleo generator uses UML models to create the mapper. Reading is done from the SQL Server and writing is done in B2MML format.

Structure of the data that is read to the interface represents the actual tables from the SQL database. The interface has a somewhat simplified version of the actual database since not every connection between the tables is required, and many values are not used at all. After the simplification, the data structure is a tree. Child nodes are stored as lists inside their respective parent nodes and some of the nodes also have attributes.

B2MML styled document is created using javax document builder that systematically writes the information stored in the interface class tree as XML. Conversions regarding timestamps are required since SQL and XML and Java differ in the way it should be written. Schema, B2MML version, namespace, etc. are added at the start of the file.

In order to compare the generated solution to something meaningful, a manual example of the mapper was created. The manual example was made with Java. The code generator was implemented with Acceleo [20]. Acceleo uses templating to create text from models.

During the implementation of the generator it became apparent that it would not be possible to design something that would closely resemble the manual example and follow the requirement to keep the generator general.

The generated mapper code does the data transformation in a completely different manner than the manually implemented one. The main reason is that in the manual example classes that have subclasses call the functions on the spot in the loop, leading to standard loop inside a loop structure that is common in programming. The most basic example of this would be *for x* that has *for y* inside it. However, the generated implementation that would do this could not be made with current design and requirements since it would not be possible to place *for y* inside the *for x* loop. The generated implementation executes *for y* separately from *for x* and later combines them. When comparing figures 17 and 18 to each other it can be seen that while there are similarities, they are different. One interesting thing is that there is a **if(true)** that segments the generated code into pieces. There reason is that when multiple loops are generated and they use the same code to generate the loops, the names of the variables are the same. By segregating the loops from each other, same names will not cause troubles.

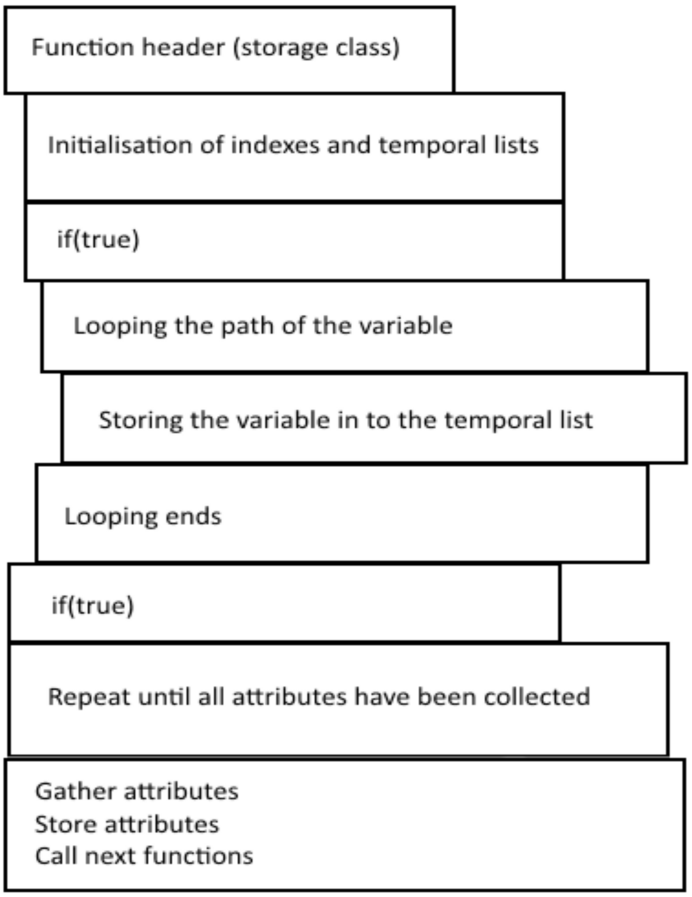

Figure 17: The structure of a function of the generated mapper. Data is stored into the given parameter. The indents represent the code hierarchy

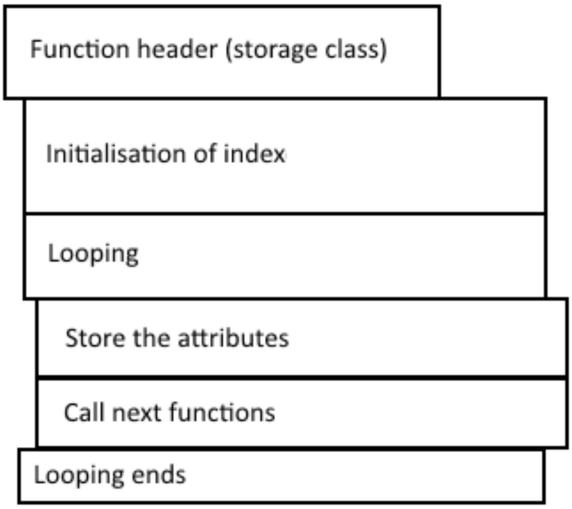

Figure 18: The structure of a function of the manual mapper. Data is stored into the given parameter. The indents represent the code hierarchy.

The generated solution collects data inside the loops and stores it into temporal lists that are created before the loops. After looping the attributes they are gathered by a piece of code and stored properly into the receiving data structure. After the data is stored, next functions are called in order to proceed. The looping is equivalent to the algorithm's collection of the mapped attributes section. The main difference between the generated and the manually written algorithm is that the generated one collects each attribute under a class separately and then combines them, while the manual one gathers them all at the same time by calling functions.

The code generator was implemented by merging small functions. It is organised into two main parts, an initialisation template and multi-use templates. The initialisation template creates functions one by one. Logic of the function creation follows the algorithm seen in the design chapter. The initialisation template also generates the starting point of the mapper code.

Multi-use templates can be divided into traversing templates, writing templates and templates that help to keep track of variables namely gatherer. Traverse templates navigate inside the tree. Some traverse templates move from class nodes to attribute nodes and from the attribute nodes to other attributes. All of these templates are simple an exist only for systematical traversing and data gathering. One traversing template finds an association and another moves to the other end of it. The amount of information collected increases with every move.

Writing templates write the loops that go trough the data and the code inside the loops. The loops form the pathway to the location of the data. Looping is done for each parent of the attribute all the way to the top of the tree. Data required for the writing has been gathered with the traversing templares. While traversing templates are simple, writing templates are complex. Data for the loops is gathered in reverse order(bottom to top). Writing in reverse causes the templates to be complex. Looping templates utilise double recursion in order to go through the data. In regular programming, this would not be a problem, however Acceleo presents its own technical challenges. Because of Acceleo uses OCL, variables cannot change values [19]. If something needs to be stored, a new variable has to be created. Therefore, simple programming styles, such as the use of flags in loops, is impossible. Because of this constraint, some complex recursion within a recursion was used to generate the code that writes the loops. Writing code inside loops is not as complex a task as the looping.

Other problems regarding the technical difficulties include the inability to use 'elseif' statements because of OCL constraints and the lack of documentation related to similar problems when using Acceleo. Most example cases that were readily available use Acceleo to write classes. In addition, the older versions of Acceleo are different than the version used in this thesis. Naturally, there might be better ways to create the code generator than the one used. Also, by allowing less general solutions, better results could be possible.

There is also the writing of the gatherer. The purpose of the gatherer is to collect the attributes under one instance of the class which is then stored in its proper place the output. Without it, there would be multiple instances of the same output class and all of them would have only one piece of the relevant information.

One important helper template group are naming helpers. They are used to create unique names for the variables and the classes. The created names are used in the mapper. Names are based on the UML model naming style. Unique names are important for the gatherer since the data is temporally stored in unique classes before the gatherer combines the data. Another important helper template collects information from neighbouring nodes. For example *Atr_cMR_MaterialUseGlb_material* is a name created from the combined information from neighbouring nodes. *Atr_cMR_MaterialUse* and *Glb_material* are the names of the neighbouring nodes and as names of the nodes are unique the created name is unique and that unique name can be used as a name of a variable without having problems with the naming.

One interesting feature that was experimented on was allowing custom functions. Function usage was implemented into the model where attributes are mapped into each other. By adding an extra module in between the mapped attributes, it was possible to write outside function usage into the generated mapper (figure 19). These functions could be used, for example, to manipulate the attributes. Before this there were manipulation commands in the model itself. Thus instead of having ”merge attributes”-command inside the attribute, it could be outsourced into a premade helper function. Moving functionality into premade functions and then calling them through the model does not break the generality requirement since the function calls are in the mapped model. This kind of outside functionality could help with creation of better generators.

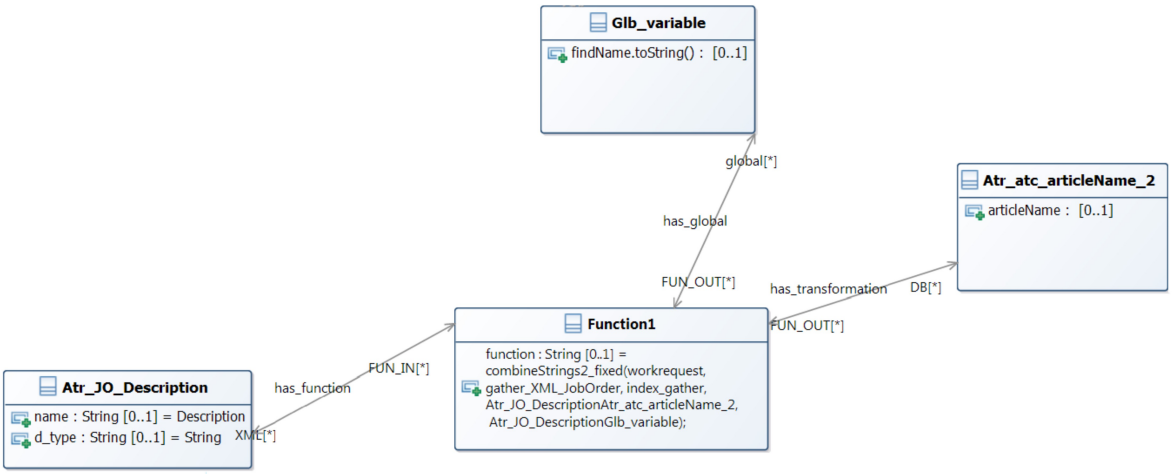

Figure 19: Custom function between mapped attributes allows new possibilities for the creation of the generator. This function merges two strings.(Unnecessary information has been removed in order to clarify the figure.)

6.2 Experimentation

The test case was a work schedule of a test project that the user of the system requires as a B2MML-styled document. A manually created mapper was used as a reference to make a comparison between the generated and the handwritten versions.

The manually implemented mapper has considerably less looping and is thus more efficient than the generated mapper. The difference in the size of the code is also considerable: the manually created mapper is about 200 lines while the generated one consists of about 700 lines. They have different solutions to the mapping since the generator that was implemented is a general solution to similar problems. With a limited amount of non-general functionality and helper functions added into the general mapper, it could be possible to modify it to produce code that resembles the manually created mapper.

Different sized trees were also tested by cutting and adding more material to the work schedule. Since the generator was tested often with different sets, it can be said that it can work with trees that have different size and shape. Therefore, the requirement for a general code generator is met in similar cases.

Custom functions for the attribute mapping were a better solution for the attribute mapping than just having the functionality in the recieving attribute. Superiority of custom functions comes from easier implementation and from the ability to use already made custom functions.

One problem with the UML model was that it was not found possible to control in which order the associations are found. This would cause situation where instead of writing ”left axel 9250” in the B2MML file, ”9250 left axel” would be written instead. If the order of combined attributes is not properly determined, it can pose problems for this kind of code generation. In order to fix it would be beneficial to have a feature in UML that allows managing the order of the associations. Technically the order could be added into the model as an attribute but doing it that way puts more strain to the developer. Other than this, the B2MML document was generated successfully.

During the testing of complex mapping, one bug was found in the tracking of the IDs of the instances. If tree branch was changed multiple times in a row, it was possible to lose the proper instance. While correct variable was found, it could be the wrong instance of the variable. The bug was ignored, since fixing it would have taken considerable amount of time and it did not affect the end result of the main test subject. There are couple of fixes to this problem. First potential fix would be to have fully unique ids for the instances. This means that when extracting data from the database the key ID values should not only follow SQL rules of uniqueness but also be unique with all other key values. While this is a lot to ask for from an existing database, it could be done with completely new databases. A second solution is that the ids are made unique when they are extracted from the database

by adding a prefix based on them. These two solutions are simple solutions because if the IDs are truly unique the tracking becomes easier. Third solution is to keep track of the whole tree branches IDs in order to find the wanted instance of the variable. This could become troublesome if the trees become really large. Writing this solution into the generator is probably the most cumbersome of the potential solutions.

7 Conclusions

The objective of this thesis was to gather knowledge about the usage of the MDE methods in the application integration of MES. This knowledge was achieved by creating an experimental setup. Different kinds of tests were then conducted with the test setup. As a proof of concept, it was seen that at least some software integration of the MES can be done with model driven approach.

The requirements were defined in Chapter 4. What the developer desired was to save workhours and create the system with less work. Based on what was learned from the experiment, it can be stated that MDE methods can eventually create systems with less work. This was learned by making big changes in the model and noticing that the code generator worked with the updated model. The real problem with saving the actual workhours is that adoption of the MDE methods requires a large amount of work before it can even be used. Where the time required to setup the MDE and create code, and the time required to manually program the system meet depends a lot on how complex the system is and how many times code generator can be re-used. But there is quite likely an advantage for MDE in large data sets that are used multiple times.

The design of the code generator allows the usage of tree shaped data structure and models with certain naming rules. Naturally, suitable interfaces are needed to read and write the data so that the code generator can handle it, however the code generator is not restricted to MES-only databases, nor is the output data restricted to B2MML only. It does not matter from where the data is taken from, or in what kind of data format it is going to be output, as long as the data is in a tree shape and it follows the naming rules. This means that the code generator introduced here can be re-used on similar cases for other systems.

Since a general MDE solution was wanted, it placed restrictions on how the data could be used. These restrictions caused worse performance as can be instantly seen when comparing the amount of the looping done to the manually programmed code. This could cause problems with applications that have performance requirements. Furthermore, the lenght of the generated code was almoust four times longer. This is not a major issue, because memory is not such an important issue anymore as it used to be in the past. However, this could be a problem for low memory devices that have complex softwares.

During the implementation of the system it was noticed that when mapping the models together, external functions may be added to the mappings to provide additional functionalities. This opens up large possibilities since all kinds of different functionality could be added just by editing the models without touching the code generator. These functions could be taken from existing function libraries thus making the job of the developer easier.

During the experimentation it was noticed that a lot of similar styled templates were used multiple times. It could be worthwhile future work to investigate if parts of the generator could be used as a part of a library that would help to create other generators. As long as the naming rules of the templates are followed, nothing prevents using them in other generators. Good examples of the templates that could be helpful for other Acceleo users are the navigation templates that are used to traverse the tree. Another good example is the loop writing templates that create the loops inside the loops. Especially the recursion solution for looping is somewhat complex. Thus if someone were to advance on this topic, creation of a tool that generates templates to common problems could be beneficial (figure 20).

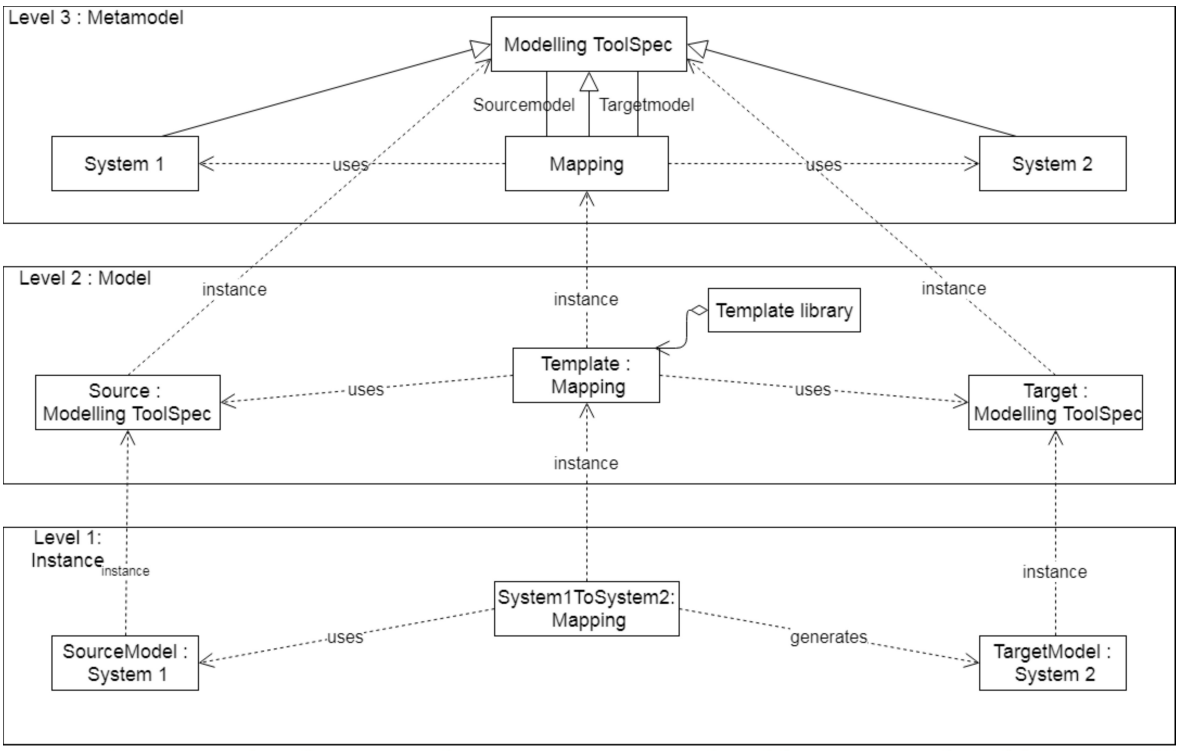

Figure 20: Example of mapping between systems with a library. Template library could ease the creation of generators and speed up the development process. Edited [26]

In addition to the template library, creation of a tool that can be used to generate UML models automatically from the dataset used could be useful. Generating the UML models from interfaces with algorithms should be advantageous if the size of the data structure is large. Writing such an algorithm should be possible for tree shaped datastuctures but it might be problematic to write one for more complex structures.

Model driven approach is likely going to keep slowly spreading since there are potential savings and industries tend to gravitate towards profits. Automating programming is the logical next step for the process that started with industrial revolution. This could eventually start applying pressure on software engineers, since more and more software can be generated, thus forcing the engineers to adapt and learn the model based approach or take more complex programming challenges since automation will replace the easier programming jobs.

References

[1] Alon Halevy, et al. Data integration: The teenage years ,VLDB 2006 - Proceedings of the 32nd International Conference on Very Large Data Bases. 9-16. 2006

[2] P. Caserta and O. Zendra, "Visualization of the Static Aspects of Software: A Survey," in IEEE Transactions on Visualization and Computer Graphics, vol. 17, no. 7, pp. 913-933, July 2011.

[3] ISA. ANSI/ISA-95.00.01-2000, Enterprise-Control System Integration, Part 1:
Models and Terminology. International Society of Automation, 2000.

[4] Kletti, Jürgen. Manufacturing Execution Systems – MES. New York: Springer, 2007. ISBN 978-3-540-49743-1.

[5] ISA. ANSI/ISA-95.00.03-2005, Enterprise-Control System Integration, Part 3: Activity
Models of Manufacturing. International Society of Automation, 2005b.

[6] Meyer Heiko, et al. Manufacturing execution systems: optimal design, planning, and deployment. New York: McGraw-Hill, 2009. ISBN 978-0-07-162383-4.

[7] Robin G. Qiu and Mengchu Zhou. Mighty MESs; State-of-the-Art and Future Manufacturing Execution Systems. Robotics & Automation Magazine, IEEE, 11(1): 1925, 2004.

[8] Linthicum, David S. Next Generation Application Integration : From Simple Information to Web Services. Boston: Addison-Wesley, 2004. ISBN 0-201-84456-7.

[9] C. Jiantao, Q. Yuanying and K. Xianguang, "Integrated System of ERP and MES Based on DHNN Evaluation," 2013 Fourth International Conference on Digital Manufacturing & Automation, Qingdao, 2013, pp. 709-712.

[10] Y. Wang, SCM/ERP/MES/PCS integration for process enterprise, Proceedings of the 29th Chinese Control Conference, Beijing, 2010, pp. 5329-5332.

[11] ISA. ANSI/ISA-95.00.05-2005, Enterprise-Control System Integration, Part 4: Object
Models and Attributes of Manufacturing Operations Management, Draft 3.
International Society of Automation, 2005a.

[12] MESA international. B2MML-BatchML-V0600-Release Notes. 2013 available: http://www.mesa.org/en/B2MML.asp

[13] Brambilla Marco, Cabot Jordi and Wimmer Manuel. Model-Driven Software Engineering in Practice. San Rafael, Calif. (1537 Fourth Street, San Rafael, CA 94901 USA): Morgan & Claypool, 2012. ISBN 978-1-60845-883-7

[14] E. Brottier, F. Fleurey, J. Steel, B. Baudry and Y. L. Traon, Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool, 2006 17th International Symposium on Software Reliability Engineering, Raleigh, NC, 2006, pp. 85-94.

[15] M. Asadi, M. Ravakhah and R. Ramsin, An MDA-Based System Development Lifecycle, 2008 Second Asia International Conference on Modelling & Simulation (AMS), Kuala Lumpur, 2008, pp. 836-842.

[16] Stahl Thomas, Markus Völter, et al. Model-Driven Software Development: Technology, Engineering, Management, England 2006, John Wiley & Sons Ltd,. ISBN:0-470-02570-0

[17] Object Management Group, OMG Unified Modeling LanguageTM (OMG UML), Superstructure, OMG, 2011

[18] Object Management Group, Meta Object Facility (MOF) Core Specification, OMG, 2006

[19] Object Management Group, OMG Object Constraint Language (OCL), OMG, 2012

[20] Acceleo, available: http://www.eclipse.org/acceleo/

[21] Eclipse Modeling Framework, available: https://www.eclipse.org/modeling/emf/

[22] Object Management Group, MOF Model to Text Transformation Language, v1.0, OMG, 2008

[23] Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, OMG, 2016.

[24] UML Designer, available: http://www.umldesigner.org/

[25] R. L. Glass, Frequently forgotten fundamental facts about software engineering, in IEEE Software, vol. 18, no. 3, pp. 112-111, May 2001.

[26] Berger S., Grossmann G., Stumptner M., Schrefl M. Metamodel-Based Information Integration at Industrial Scale. In: Petriu D.C., Rouquette N., Haugen Ø. (eds) Model Driven Engineering Languages and Systems. MODELS 2010. Lecture Notes in Computer Science, vol 6395. Springer, Berlin, Heidelberg, 2010